

Fully Abstract Trace Semantics for Low-level Isolation Mechanisms

Marco Patrignani^{*} Dave Clarke
iMinds-DistriNet, Dept. Computer Science, KU Leuven
{first.last}@cs.kuleuven.be

ABSTRACT

Many software systems adopt isolation mechanisms of modern processors as software security building blocks. Reasoning about these building blocks means reasoning about elaborate assembly code, which can be very complex due to the loose structure of the code. A way to overcome this complexity is giving the code a more structured semantics. This paper presents one such semantics, namely a *fully abstract* trace semantics, for an assembly language enhanced with protection mechanisms of modern processors. The trace semantics represents the behaviour of protected assembly code with simple abstractions, unburdened by low-level details, at the maximum degree of precision. Additionally, it captures the capabilities of attackers to protected software and simplifies providing a secure compiler targeting that language.

Categories and Subject Descriptors

D.2.4 [Software/Program Verification]: Formal models

1. INTRODUCTION

Many processors provide isolation mechanisms as software security building blocks. These are used to withstand low-level attackers who, through injected assembly code, can read the whole memory space and access secrets in memory, violate integrity constraints and so on. With isolation mechanisms, attackers cannot directly violate security properties of isolated software since the isolated memory is not accessible to them. Examples of these protection mechanisms are fine grained, program counter-based memory access control mechanism [4, 8, 9, 12, 17, 18, 19], which enforce security properties at process or lower levels (Ring-1). With this mechanism, the software to be secured is placed in a protected memory partition that shields it from the surrounding, potentially malicious code. The malicious code cannot read nor write the protected memory; it can only jump to specific addresses in protected memory to call functions of

the protected code. This protection mechanism makes software more resilient against low-level attackers. However, this does not prevent an attacker from interacting with the protected code and violating security properties.

To enable reasoning about the behaviour of isolated software, this paper firstly introduces an assembly language enhanced with a fine grained, program counter-based memory access control. This protection mechanism was chosen due to its application in secure compilation [2, 14] and to its recent industrial implementation: the Intel SGX [9]. Then, this paper explores the design space of trace semantics and presents two different *fully abstract* [15] trace semantics for the protected assembly code. The paper provides a general proof strategy where full abstraction of the trace semantics can be proven simply. To the best of the authors' knowledge, this is the first such result for an assembly language extended with the said protection mechanism. Moreover, the results seem applicable to other isolation mechanisms.

The fully abstract trace semantics provides a number of benefits. Trace semantics uses simple abstractions to represent the behaviour of protected assembly code, unburdened by low-level details, at the maximum degree of precision. So, dually, it models the behaviour of an attacker to the protected code, since it captures precisely the capabilities of that attacker. Secondly, without the trace semantics, the behaviour of protected assembly code (and thus of an attacker) is given in terms of low-level contexts. While being precise, low-level contexts are notoriously difficult to reason about, since they offer no inductive nor coinductive structure. The fully abstract trace semantics allows low-level contexts to be disregarded, since low-level contexts and traces are proven to be equally precise. Finally, the field of secure compilation benefits from the trace semantics. Recently, Agten *et al.* [2] and Patrignani *et al.* [14] presented secure compilers to assembly code enhanced with the aforementioned protection mechanism. Both results assume that the assembly language has a fully abstract trace semantics, thus the results of this paper strengthen those secure compilation claims.

In summary, the contributions of this paper are:

- the formalisation of an assembly language modelling an architecture enhanced with a fine grained, program counter-based memory access control;
- two different trace semantics for that low-level model;
- the proof that both trace semantics are fully abstract in a general proof strategy.

Limitations of this work are twofold. Firstly, the trace semantics cannot express side-channel attacks. Secondly, the

^{*}Marco Patrignani holds a Ph.D. fellowship from the Research Foundation Flanders (FWO).

formalisation does not consider details of the architecture such as caches; yet this is often assumed [2, 11, 14, 16].

The paper is organised as follows. [Section 2](#) introduces background notions for the work. [Section 3](#) formalises syntax and operational semantics of the assembly language. [Section 4](#) describes two trace semantics for the language. [Section 5](#) contains the proof of full abstraction of both trace semantics. [Section 6](#) describes related work. [Section 7](#) discusses future works and concludes.

2. BACKGROUND NOTIONS

This section describes the memory access control mechanism. Then, it describes a fully abstract trace semantics for protected assembly code by turning an imprecise representation of the behaviour of the code into a precise one.

2.1 The Protection Mechanism, Informally

The assembly language is enhanced with an isolation mechanism: a fine-grained, program counter-based memory access control mechanism [4, 8, 12, 17, 18, 19]. We review this mechanism from the work of Strackx and Piessens [18]. However, the techniques presented in this paper can be easily adapted to reasoning about other isolation mechanisms [4, 8, 17]. The protection mechanism provides a secure environment for running code that must be protected from the code it interacts with. This mechanism assumes that the memory is logically divided into a *protected* and an *unprotected* partition. The protected partition is further divided into a *code* and a *data* section. The code section contains a variable number of *entry points*: the only addresses to which instructions in unprotected memory can jump and execute. The data section is accessible only from the protected partition. For the sake of simplicity, code in protected memory cannot read the unprotected one; as discussed in [Section 2.2](#) below. Based on the location of the program counter, instructions that violate the access control policy are ruled out. The table below summarises the access control model enforced by the protection mechanism.

From \ To	Protected			Unprotected
	Entry Point	Code	Data	
Protected	r x	r x	r w	w x
Unprotected	x			r w x

2.2 Devising Fully Abstract Trace Semantics

Following is the notation used in all code examples throughout this paper. Every instruction in the code is preceded by the address where it is located. Call the left program P_L and the right one P_R . When P_L and P_R are executed, they are placed in a protected memory partition spanning from address 100 to 200, with a single entry point at address 100.

EXAMPLE 1.

1	100	sub r ₀ r ₁	1	100	sub r ₀ r ₁
2	101	movi r ₃ 106	2	101	movi r ₃ 106
3	102	j1 r ₃	3	102	j1 r ₃
4	103	movi r ₃ 10	4	103	movi r ₃ 10
5	104	movs r ₃ r ₄	5	104	movs r ₃ r ₅
6	105	call r ₂	6	105	call r ₂
7	106	movi r ₁₁ 41	7	106	movi r ₁₁ 42
8	107	movi r ₀ 0	8	107	movi r ₀ 0
9	108	ret	9	108	ret

Both P_L and P_R assign to r_0 the result of $r_0 - r_1$ (line 1). If the result of the operation is not less than 0 (line 3), they respectively write the contents of r_4 and r_5 at the unprotected

address 10 (lines 4,5) and call a function whose address is stored in r_2 (line 6). Otherwise, they assign different values to r_{11} (line 7) and return 0 (lines 8,9).

In order to describe the behaviour of the protected code of [Example 1](#), this paper defines a traces semantics for it. The trace semantics gives a simple characterisation of the behaviour of that code as a set of sequences of labels. In this paper, trace semantics are devised to capture the execution of a *protected* program, which is a program allocated in the protected memory partition. The execution of a program is the evaluation of a sequence of instructions; in order to abstract over instructions, not all of them generate a label in a trace. Labels are generated only by **call** and **ret** instructions. If they are executed by unprotected code, they are named *calls* and *returnbacks*, if they are executed by protected code they are named *callbacks* and *returns* [2, 14]. Following is the syntax of labels.

$$L ::= a \mid \tau \quad a ::= g? \mid g! \quad g ::= \text{call } p(\bar{v}) \mid \text{ret } v$$

A label L can be an observable action a or a non-observable action τ . Decorations $?$ and $!$ indicate the direction of the observable action: from unprotected to protected code ($?$) or vice-versa ($!$). Address p is an address in memory, \bar{v} is a list of the contents of all registers and v indicates the contents of register r_0 .

Informally, a trace semantics is fully abstract when its labels capture all that is being communicated between the protected and the unprotected code. The trace semantics hinted at above is not fully abstract due to a number of subtleties, as highlighted by the following traces for the code of [Example 1](#). Omitted details are indicated using \dots ; use \cdot to separate actions of the same trace.

$$\begin{aligned} \bar{a}_1 &= \text{call } 100(1, 2, \dots)? \cdot \text{ret } 0! \\ \bar{a}_2 &= \text{call } 100(2, 1, 40, \dots)? \cdot \text{call } 40(\dots)! \end{aligned}$$

Traces \bar{a}_1 and \bar{a}_2 are generated by both P_L and P_R . Trace \bar{a}_1 does not capture the different value stored in r_{11} (line 7), which, even if it is not the returned value of the function, still constitutes an observable difference between P_L and P_R . Trace \bar{a}_2 does not capture the different value written at address 10 (line 5), which also constitutes a observable difference between P_L and P_R . Since \bar{a}_1 and \bar{a}_2 do not capture the observable differences between P_L and P_R , the trace semantics fails to be fully abstract.

As Curien stated [3], two ways to achieve full abstraction for a trace semantics exist. The first is to modify the labels so that they capture more precisely what is communicated between protected and unprotected code. In this case, labels should capture the values of all registers and what protected code writes in unprotected memory. The second is to change the operational semantics, to restrict what is communicated to what is captured by the labels. This is achieved by restricting the ways in which communication is performed, e.g. by preventing writeouts. Both approaches are presented in [Section 4](#); they rely on an assembly language and on its operational semantics which are formalised in [Section 3](#).

3. LANGUAGE FORMALISATION

This section formalises the syntax, operational semantics and contextual equivalence of an assembly language.

3.1 Syntax

The language is run on an architecture that models a Von Neumann machine consisting of a program counter p , a register file r , a flags register f and memory space m . The program counter indicates the address of the instruction that is executed. The register file contains 12 general purpose registers R_0 to R_{11} and a stack pointer register SP , which contains the address of the top of the current call stack. The flags register contains a zero flag ZF and a sign flag SF , which are set or cleared by arithmetic instructions and are used by branching instructions. For the sake of simplicity, assume the architecture targeted by the language works with ℓ bit-long words, where ℓ is a power of 2. This allows the formalisation presented to scale to architectures with words of different sizes.

Words	$w ::= i \in \mathcal{I}$ $v \in \mathcal{V}$	Memories	$m ::= \emptyset$ $m; a \mapsto w$
Empty word	$\mathbf{0} ::= 0^\ell$	Numbers	$n ::= n \in \mathbb{N}$
Addresses	$a \in 0..2^\ell - 1$	Programs	$P ::= (m, s)$
Values	$\mathcal{V} ::= \{v \mid v = [0 \text{ or } 1]^\ell \wedge \forall i \in \mathcal{I}. v \neq i\}$		
Memory descriptors	$s ::= (a_b, n_c, n_d, n)$		

Figure 1: Assembly language formalisation.

Fig. 1 presents elements of the formalisation. Words w are either instructions i or values v . Values v are elements of the set \mathcal{V} , they are sequences of bits 0 or 1 of length ℓ that are different from the encoding of instructions. The explanation of why values are encoded in this way is delayed to Example 3 in Section 4.1. Instructions i are elements of the set \mathcal{I} and define the programming language executed on the architecture (Fig. 2). The empty word $\mathbf{0}$ is a sequence of 0's

<code>movl r_d r_s</code>	Load the word from the memory address in register r_s into register r_d .
<code>movs r_d r_s</code>	Store the contents of register r_s at the address found in register r_d .
<code>movi r_d k</code>	Load the constant value k into register r_d . Note that $k < 2^\ell$.
<code>add r_d r_s</code>	Write $r_d + r_s \bmod 2^\ell$ into register r_d and set the ZF flag accordingly.
<code>sub r_d r_s</code>	Write $r_d - r_s \bmod 2^\ell$ into register r_d and set both the ZF and the SF flags accordingly.
<code>cmp r_1 r_2</code>	Calculate $r_1 - r_2$ and set both the ZF and the SF flags accordingly.
<code>jmp r_1</code>	Jump to the address located in register r_1 .
<code>je r_1</code>	If the ZF flag is set, jump to the address in register r_1 .
<code>j1 r_1</code>	If the SF flag is set, jump to the address in register r_1 .
<code>call r_1</code>	Push the value of the program counter +1 onto the stack and jump to the address in register r_1 .
<code>ret</code>	Pop a value from the stack and jump to the popped location.
<code>halt</code>	Stop the execution with the result in register r_0 .

Figure 2: Instruction set \mathcal{I} .

whose length is based on the architecture being considered. Addresses a are natural numbers, ranging from 0 to $2^\ell - 1$. Memories m are maps from addresses to words. Memory access, denoted as $m(a)$, is defined as follows: $m(a) = w$ if $a \mapsto w \in m$; it is undefined otherwise. Define the domain of a memory as $\text{dom}(m) = \{a \mid a \mapsto w \in m\}$. If two memories m and m' have disjoint domains, they can be merged in

another memory. Formally, if $\text{dom}(m) \cap \text{dom}(m') = \emptyset$, then $m + m' = \{a \mapsto w \mid a \mapsto w \in m \text{ or } a \mapsto w \in m'\}$. Memory descriptors s are quadruples: (a_b, n_c, n_d, n) that formalise the concepts of Section 2.1: a_b is the address where the protected memory partition starts, n_c and n_d are the sizes (in number of addresses) of the code and data section respectively and n is the number of entry points. Entry points are allocated starting from the base address a_b . Each entry point is \mathcal{N}_e words long. Assume the entry points do not overflow the protected code section, thus the constraint $n \cdot \mathcal{N}_e < n_c$ holds for the all memory descriptors. Programs P are pairs of a memory m and a memory descriptor s .

3.2 Operational Semantics

Before introducing the semantics, a number of auxiliary notions are defined.

Fig. 3 defines the access control enforcement rules informally presented in Section 2.1. Read judgments $s \vdash \text{predicate}(a, b, \dots)$ as: “according to s , predicate holds for addresses a, b, \dots ”. Assume $s \equiv (a_b, n_c, n_d, n)$.

(Aux-protected)	(Aux-unprotected1)
$\frac{a_b \leq p < (a_b + n_c + n_d)}{s \vdash \text{protected}(p)}$	$\frac{p < a_b}{s \vdash \text{unprotected}(p)}$
(Aux-unprotected2)	(Aux-returnEntry)
$\frac{(a_b + n_c + n_d) \leq p}{s \vdash \text{unprotected}(p)}$	$\frac{p = a_b + (n - 1) \cdot \mathcal{N}_e}{s \vdash \text{returnEntryPoint}(p)}$
(Aux-entryPoint)	(Aux-data)
$\frac{p = a_b + m \cdot \mathcal{N}_e}{m \in \mathbb{N} \quad m < n}$	$\frac{(a_b + n_c) \leq p}{p < (a_b + n_c + n_d)}$
$s \vdash \text{entryPoint}(p)$	$s \vdash \text{data}(p)$
(Aux-read-1)	(Aux-read-2)
$\frac{s \vdash \text{protected}(p) \quad s \vdash \text{protected}(a)}{s \vdash \text{readAllowed}(p, a)}$	$\frac{s \vdash \text{unprotected}(p) \quad s \vdash \text{unprotected}(a)}{s \vdash \text{readAllowed}(p, a)}$
(Aux-write-1)	(Aux-write-2)
$\frac{s \vdash \text{unprotected}(a)}{s \vdash \text{writeAllowed}(p, a)}$	$\frac{s \vdash \text{protected}(p) \quad s \vdash \text{data}(a)}{s \vdash \text{writeAllowed}(p, a)}$
(Aux-entry)	(Aux-return)
$\frac{s \vdash \text{unprotected}(p) \quad s \vdash \text{entryPoint}(p')}{s \vdash \text{entryJump}(p, p')}$	$\frac{s \vdash \text{protected}(p) \quad s \vdash \text{unprotected}(p')}{s \vdash \text{exitJump}(p, p')}$
(Aux-internal)	(Aux-external)
$\frac{s \vdash \text{protected}(p) \quad s \vdash \text{protected}(p') \quad s \not\vdash \text{data}(p')}{s \vdash \text{intJump}(p, p')}$	$\frac{s \vdash \text{unprotected}(p) \quad s \vdash \text{unprotected}(p')}{s \vdash \text{extJump}(p, p')}$

Figure 3: Access control enforcement rules.

Define functions $m_{\text{sec}}(m, s)$ and $m_{\text{ext}}(m, s)$, which return the protected and unprotected parts of a memory m according to the descriptor s , respectively as: $m_{\text{sec}}(m, s) = \{a \mapsto w \mid a \mapsto w \in m, s \vdash \text{protected}(a)\}$ and $m_{\text{ext}}(m, s) = \{a \mapsto w \mid a \mapsto w \in m, s \vdash \text{unprotected}(a)\}$.

In the semantics there are two call stacks, one for the protected code, called the *secure stack*, and one for the unprotected code, called the *insecure stack*. Each stack is preceded by a word containing the location of the current head of the stack, let SP_{sec} and SP_{ext} indicate the address of the secure and insecure stack respectively. When the current

stack is changed in the semantics, the stack pointer register SP is initialised to the right address using SP_{sec} and SP_{ext} . Given a memory descriptor $s = (a_b, n_c, n_d, n)$, the secure stack starts at the beginning of the protected data section and the insecure stack starts at the end of the protected memory partition. Thus $\text{SP}_{\text{sec}} = (a_b + n_c)$ and, initially, $\text{SP}_{\text{sec}} \mapsto (a_b + n_c + 1)$; analogously, $\text{SP}_{\text{ext}} = (a_b + n_c + n_d)$ and, initially, $\text{SP}_{\text{ext}} \mapsto (a_b + n_c + n_d + 1)$. Call and return instructions are responsible of setting the SP register to the correct address when crossing boundaries between protected and unprotected memory. The value of the program counter is pushed onto the stack by a **call** instruction (the stack grows down), while a **ret** instruction pops one address from the top of the stack and jumps to that location. Updating the stack pointer SP is performed by the auxiliary function **setStack** (Fig. 4). In the rules, notation $m[a \mapsto w]$ indicates that memory m is updated to a new one that is equal to m except that the value stored at address a is w . Notation $r[R \mapsto w]$ indicates that the register file r is updated to a new one that is equal to r except that the value stored in register R is w . Notation $r(R)$ indicates the value contained in register R in register file r . Given a jump between

$$\begin{array}{c}
\text{(Stack-out-to-in)} \\
\frac{s \vdash \text{entryJump}(p, p') \quad m' = m[\text{SP}_{\text{ext}} \mapsto r(\text{SP})] \quad r' = r[\text{SP} \mapsto m(\text{SP}_{\text{sec}})] \quad s \vdash \text{unprotected } r(\text{SP}) \quad s \vdash \text{protected } r'(\text{SP})}{p, r, m, s \vdash \text{setStack} \searrow p', r', m'} \\
\text{(Stack-in-to-out)} \\
\frac{s \vdash \text{exitJump}(p, p') \quad m' = m[\text{SP}_{\text{sec}} \mapsto r(\text{SP})] \quad r' = r[\text{SP} \mapsto m(\text{SP}_{\text{ext}})] \quad s \vdash \text{protected } r(\text{SP}) \quad s \vdash \text{unprotected } r'(\text{SP})}{p, r, m, s \vdash \text{setStack} \searrow p', r', m'} \\
\text{(Stack-no-change-i)} \\
\frac{s \vdash \text{intJump}(p, p') \quad s \vdash \text{protected } r(\text{SP})}{p, r, m, s \vdash \text{setStack} \searrow p', r, m} \\
\text{(Stack-no-change-e)} \\
\frac{s \vdash \text{extJump}(p, p') \quad s \vdash \text{unprotected } r(\text{SP})}{p, r, m, s \vdash \text{setStack} \searrow p', r, m}
\end{array}$$

Figure 4: Stack switch enforcement rules.

addresses p and p' , the stack switch rules produce a new register file r' and a new memory m' based on old ones r and m . The memory is updated to store the top of the current stack, located in SP, in the address storing the top of the current stack: SP_{sec} or SP_{ext} . When the stack is changed, the register file is updated to initialise SP to the top of the right stack: the address stored at SP_{sec} or SP_{ext} .

The operational semantics is a small step semantics that describes how each instruction of the language transforms an execution state into a new one. Thus, the operational semantics handles programs in the whole memory: both the protected and unprotected partitions.

DEFINITION 1 (EXECUTION STATE). *An execution state, denoted as Ω , is a quintuple $\Omega = (p, r, f, m, s)$, where p is a program counter, r is a register file, f is a flags register, m is a memory and s is a memory descriptor.*

Given $\Omega = (p, r, f, m, s)$, let $[\Omega]$ be the state $(p, r, f, m_{\text{sec}}(m, s), s)$ and $[\Omega']$ be the state $(p, r, f, m_{\text{ext}}(m, s), s)$. Relations $\xrightarrow{i} \subseteq [\Omega] \times [\Omega]$ and $\xrightarrow{e} \subseteq [\Omega] \times [\Omega]$ describe the evaluation of instructions that only affect the protected and unprotected parts of memory respectively. Fig. 5 presents the rules for

\xrightarrow{i} , rules for \xrightarrow{e} are obtained by replacing an **intJump** assumption with an **extJump** one. Let $m(p) = \text{inst}$ denote that **inst** is the word allocated in $m(p)$, where $\text{inst} \in \mathcal{I}$. To capture termination, the program counter is set to -1 whenever the **halt** instruction is encountered. This way, no progress can be made, as $m(-1)$ does not return a valid instruction: the program is in a stuck state.

DEFINITION 2 (STUCK STATE). *A state $\Omega = (p, r, f, m, s)$ is stuck, denoted as Ω^\perp , when the program counter does not point to a valid instruction: $m(p) \notin \mathcal{I}$.*

The operational semantics of the language is a binary relation over states $\rightarrow \subseteq \Omega \times \Omega$ defined by the rules of Fig. 6. Rules **Eval-callback** and **Eval-returnback** ensure that the ad-

$$\begin{array}{c}
\frac{\text{(Eval-protected)} \quad [\Omega] \xrightarrow{i} [\Omega']}{\Omega \rightarrow \Omega'} \quad \frac{\text{(Eval-unprotected)} \quad [\Omega] \xrightarrow{e} [\Omega']}{\Omega \rightarrow \Omega'} \\
\text{(Eval-movs-out)} \\
\frac{m(p) = (\text{movs } r_d \ r_s) \quad s \vdash \text{intJump}(p, p+1) \quad s \vdash \text{writeAllowed}(p, r(r_d)) \quad m' = m[r(r_d) \mapsto r(r_s)] \quad r(r_s) \in \mathcal{V}}{(p, r, f, m, s) \rightarrow (p+1, r, f, m', s)} \\
\text{(Eval-callback)} \\
\frac{m(p) = (\text{call } r_d) \quad p' = m(r(r_d)) \quad s \vdash \text{exitJump}(p, p') \quad r' = r[\text{SP} \mapsto r(\text{SP}) + 1] \quad m' = m[r(\text{SP}) \mapsto p+1] \quad p, r', m', s \vdash \text{setStack} \searrow p', r'', m'' \quad r''' = r''[\text{SP} \mapsto r''(\text{SP}) + 1] \quad m''' = m'[r'''(\text{SP}) \mapsto A_{rb}]}{(p, r, f, m, s) \rightarrow (p', r''', f, m''', s)} \\
\text{(Eval-call)} \\
\frac{m(p) = (\text{call } r_d) \quad p' = m(r(r_d)) \quad s \vdash \text{entryJump}(p, p') \quad p, r, m, s \vdash \text{setStack} \searrow p', r', m' \quad r'' = r'[\text{SP} \mapsto r'(\text{SP}) + 1] \quad m'' = m'[r''(\text{SP}) \mapsto p+1]}{(p, r, f, m, s) \rightarrow (p', r'', f, m'', s)} \\
\text{(Eval-returnback)} \\
\frac{m(p) = (\text{ret}) \quad p' = m(r(\text{SP})) = A_{rb} \quad s \vdash \text{entryJump}(p, p') \quad r' = r[\text{SP} \mapsto r(\text{SP}) - 1] \quad p, r', m, s \vdash \text{setStack} \searrow p', r'', m'}{(p, r, f, m, s) \rightarrow (p', r'', f, m', s)} \\
\text{(Eval-return)} \\
\frac{m(p) = (\text{ret}) \quad p' = m(r(\text{SP})) \quad s \vdash \text{exitJump}(p, p') \quad r' = r[\text{SP} \mapsto r(\text{SP}) - 1] \quad p, r', m, s \vdash \text{setStack} \searrow p', r'', m'}{(p, r, f, m, s) \rightarrow (p', r'', f, m', s)}
\end{array}$$

Figure 6: Operational semantics of whole programs.

dress to be followed after a callback is stored in the secure stack and that the address returnback entry point A_{rb} is pushed on the insecure stack. Thus the unprotected code always jumps to the returnback entry point when returning from a callback. Code located at the returnback entry point must contain a **ret** instruction in order to correctly resume the execution. Rule **Eval-movs-out** ensures that values written in unprotected memory are not instructions.

The transitive closure of relation \rightarrow is indicated with \rightarrow^* . A state Ω performing n reduction steps is indicated as $\Omega \rightarrow^n \Omega'$. The evaluation of program P is a sequence of steps that takes the initial state of P to another state.

DEFINITION 3 (INITIAL STATE). *The initial state of a program (m, s) , denoted as $\Omega_0(m, s)$, is the state (p_0, r_0, f_0, m, s) , where $s = (a_b, n_c, n_d, n)$, $p_0 = (a_b + n_c + n_d + 2)$, $r_0 = [\text{SP} \mapsto m(\text{SP}_{\text{ext}}); r_i \mapsto 0 \text{ } i=0..11]$, and $f_0 = [\text{ZF} \mapsto 0; \text{SF} \mapsto 0]$.*

The evaluation of P terminates if $\Omega_0(P) \rightarrow^* \Omega^\perp$; the result is stored in r_0 . If the evaluation of program P does not

$\frac{\begin{array}{l} \text{(Eval-movl)} \\ m(p) = (\text{movl } r_d \ r_s) \\ s \vdash \text{intJump}(p, p+1) \\ s \vdash \text{readAllowed}(p, r(r_s)) \\ r' = r[r_d \mapsto m(r(r_s))] \end{array}}{(p, r, f, m, s) \xrightarrow{i} (p+1, r', f, m, s)}$	$\frac{\begin{array}{l} \text{(Eval-movs)} \\ m(p) = (\text{movs } r_d \ r_s) \\ s \vdash \text{intJump}(p, p+1) \\ s \vdash \text{writeAllowed}(p, r(r_d)) \\ m' = m[r(r_d) \mapsto r(r_s)] \end{array}}{(p, r, f, m, s) \xrightarrow{i} (p+1, r, f, m', s)}$	$\frac{\begin{array}{l} \text{(Eval-movi)} \\ m(p) = (\text{movi } r_d \ i) \\ s \vdash \text{intJump}(p, p+1) \\ r' = r[r_d \mapsto i] \end{array}}{(p, r, f, m, s) \xrightarrow{i} (p+1, r', f, m, s)}$
$\frac{\begin{array}{l} \text{(Eval-compare)} \\ m(p) = (\text{cmp } r_1 \ r_2) \quad s \vdash \text{intJump}(p, p+1) \\ f' = f[\text{ZF} \mapsto (r_1 == r_2); \quad \text{SF} \mapsto (r_1 < r_2)] \end{array}}{(p, r, f, m, s) \xrightarrow{i} (p+1, r, f', m, s)}$	$\frac{\begin{array}{l} \text{(Eval-add)} \\ m(p) = (\text{add } r_d \ r_s) \quad s \vdash \text{intJump}(p, p+1) \\ v = (r(r_d) + r(r_s)) \% 2^\ell \quad r' = r[r_d \mapsto v] \quad f' = f[\text{ZF} \mapsto (v == 0)] \end{array}}{(p, r, f, m, s) \xrightarrow{i} (p+1, r', f', m, s)}$	
$\frac{\begin{array}{l} \text{(Eval-sub)} \\ m(p) = (\text{sub } r_d \ r_s) \\ s \vdash \text{intJump}(p, p+1) \\ v = (r(r_d) - r(r_s)) \% 2^\ell \quad r' = r[r_d \mapsto v] \\ f' = f[\text{ZF} \mapsto (v == 0); \\ \text{SF} \mapsto (r(r_d) - r(r_s) < 0)] \end{array}}{(p, r, f, m, s) \xrightarrow{i} (p+1, r', f', m, s)}$	$\frac{\begin{array}{l} \text{(Eval-function-call)} \\ m(p) = (\text{call } r_d) \quad p' = r(r_d) \\ s \vdash \text{intJump}(p, p') \\ p, r, m, s \vdash \text{setStack} \searrow p', r', m' \\ r'' = r'[\text{SP} \mapsto r(\text{SP}) + 1] \\ m'' = m'[r''(\text{SP}) \mapsto p+1] \end{array}}{(p, r, f, m, s) \xrightarrow{i} (p', r'', f, m'', s)}$	$\frac{\begin{array}{l} \text{(Eval-function-ret)} \\ m(p) = (\text{ret}) \quad p' = r(\text{SP}) \\ s \vdash \text{intJump}(p, p') \\ r' = r[\text{SP} \mapsto r(\text{SP}) - 1] \\ p, r', m, s \vdash \text{setStack} \searrow p', r'', m' \end{array}}{(p, r, f, m, s) \xrightarrow{i} (p', r'', f, m', s)}$
$\frac{\begin{array}{l} \text{(Eval-je-true)} \\ m(p) = (\text{je } r_1) \quad f(\text{ZF}) == 1 \\ p' = r(r_1) \quad s \vdash \text{intJump}(p, p') \end{array}}{(p, r, f, m, s) \xrightarrow{i} (p', r, f, m, s)}$	$\frac{\begin{array}{l} \text{(Eval-jl-true)} \\ m(p) = (\text{jl } r_1) \quad f(\text{SF}) == 1 \\ p' = r(r_1) \quad s \vdash \text{intJump}(p, p') \end{array}}{(p, r, f, m, s) \xrightarrow{i} (p', r, f, m, s)}$	$\frac{\begin{array}{l} \text{(Eval-je-false)} \\ m(p) = (\text{je } r_1) \quad f(\text{ZF}) == 0 \\ s \vdash \text{intJump}(p, p+1) \end{array}}{(p, r, f, m, s) \xrightarrow{i} (p+1, r, f, m, s)}$
$\frac{\begin{array}{l} \text{(Eval-jl-false)} \\ m(p) = (\text{jl } r_1) \quad f(\text{SF}) == 0 \\ s \vdash \text{intJump}(p, p+1) \end{array}}{(p, r, f, m, s) \xrightarrow{i} (p+1, r, f, m, s)}$	$\frac{\begin{array}{l} \text{(Eval-jump)} \\ m(p) = (\text{jmp } r_d) \quad p' = r(r_d) \quad s \vdash \text{intJump}(p, p') \end{array}}{(p, r, f, m, s) \xrightarrow{i} (p', r, f, m, s)}$	$\frac{\begin{array}{l} \text{(Eval-halt)} \\ m(p) = (\text{halt}) \end{array}}{(p, r, f, m, s) \xrightarrow{i} (-1, r, f, m, s)}$

Figure 5: Operational semantics of instructions in the protected memory partition.

terminate, P diverges. A program P diverges, denoted as $P \uparrow$, if it executes an unbounded number of reduction steps. Formally: $P \uparrow$ if $\forall n \in \mathbb{N}, \exists \Omega'. \Omega_0(P) \rightarrow^n \Omega'$.

3.3 Contextual Equivalence

Contextual equivalence relates two programs that cannot be distinguished by any third program interacting with them [15]. This notion relies on the concept of contexts, which is introduced before presenting the equivalence itself.

Since we consider programs P that are placed in protected memory and interact with arbitrary unprotected code, contexts model that unprotected code. Thus for any descriptor s , contexts \mathbb{M} are partial memories with a hole: $\mathbb{M} = m[\cdot]$, where all addresses of \mathbb{M} are unprotected. Formally, given $s, \forall a \in \text{dom}(\mathbb{M}), s \vdash \text{unprotected}(a)$. The hole models the possibility to combine a program P with the memory \mathbb{M} iff they are compatible, denoted as $P \cap \mathbb{M}$, thus if the memories of P and \mathbb{M} have disjoint domains. Let $\text{dom}(\mathbb{M}) = \text{dom}(m)$ if $\mathbb{M} = m[\cdot]$; formally, $P \cap \mathbb{M}$ if $P = (m', s)$ and $\text{dom}(m') \cap \text{dom}(\mathbb{M}) = \emptyset$. If P and \mathbb{M} are compatible, the hole of \mathbb{M} can be filled with P in order to model interaction between P and \mathbb{M} . Formally, if $P \cap \mathbb{M}$ then $\mathbb{M}[(m', s)] = (m' + m, s)$.

Programs P_1 and P_2 are contextually equivalent, denoted as $P_1 \simeq P_2$, when, for *all* contexts they interact with, P_1 diverges if and only if P_2 also diverges.

DEFINITION 4 (CONTEXTUAL EQUIVALENCE). $P_1 \simeq P_2$ if $\forall \mathbb{M}. P_1 \cap \mathbb{M} \wedge \mathbb{M}[P_1] \uparrow \iff P_2 \cap \mathbb{M} \wedge \mathbb{M}[P_2] \uparrow$.

An implication of this definition is that for P_1 and P_2 to be contextually equivalent they must have the same memory descriptor. For the sake of simplicity we will always assume the compatibility of a program and the context it is plugged in, shortening the above definition to: $P_1 \simeq P_2$ if $\forall \mathbb{M}. \mathbb{M}[P_1] \uparrow \iff \mathbb{M}[P_2] \uparrow$.

EXAMPLE 2 (CONTEXTUALLY EQUIVALENT PROGRAMS).

The following P_L and P_R write the values of r_1 and r_2 respectively at the protected address 150 (line 2) and then return 0 (line 3). Recall that the protected memory partition spans from address 100 to 200, with one entry point at address 100.

1	100	movi r_0 150
2	101	movs r_0 r_1
3	102	movi r_0 0
4	103	ret

1	100	movi r_0 150
2	101	movs r_0 r_2
3	102	movi r_0 0
4	103	ret

The only difference between P_L and P_R is in the value stored at address 150. However, an unprotected program cannot read that value. Since that value does not affect the computation of P_L or P_R or the unprotected code, P_L and P_R are contextually equivalent.

Having defined the assembly language and its operational semantics, the paper introduces the two different trace semantics. Trace equivalence is also introduced, it will be proven the same as contextual equivalence in Section 5.

4. TRACE SEMANTICS

This section gives two different trace semantics for protected programs. The differences stem from the different ways to achieve full abstraction pointed out by Curien [3]. The first, Tr^L , possesses more expressive labels, while the second, Tr^S , relies on changes to the semantics of programs. Both are proven to be fully abstract w.r.t. the appropriate operational semantics in Section 5. Finally, this section defines when two programs are trace equivalent.

As for the operational semantics, a notion of execution states is required for the trace semantics as well. Execution states for Tr^L and Tr^S , denoted as Θ , are the same as Ω except that Θ do not deal with the whole memory but just with

its protected partition. So, the memory m of (p, r, f, m, s) spans only the protected memory partition indicated by s . Additionally, Θ can be $(\text{unknown}, m, s)$, an unknown state modelling code executing in unprotected memory.

DEFINITION 5 (INITIAL STATE FOR TRACES). *The initial state for traces of a program (m, s) , denoted as $\Theta_0(m, s)$, is the state $(\text{unknown}, m, s)$.*

4.1 Tr^L : Expressive Labels

Below are the labels exhibited by the Tr^L traces semantics.

$$\begin{aligned} \Lambda &::= \alpha \mid \tau_i & \alpha &::= \surd \mid \gamma? \mid \delta! \\ \gamma &::= (r; f) \text{call } a \mid (r; f) \text{ret } a & \delta &::= \text{write}(a, v). \delta \mid \gamma \end{aligned}$$

A label Λ can be either an observable action α or a non-observable action τ_i . Action τ_i indicates the unobservable action occurred in protected memory. Observable actions include a tick \surd indicating that the evaluation has terminated. Additionally, observable actions are function calls or returns to a certain address a , combined with the registers r and flags f . Registers and flags are in the labels as they convey information on the behaviour of programs. Observable, !-decorated actions can be prefixed by a number of write-outs of a value v to address a . They are only !-decorated since the protection mechanism forbids writes from unprotected to protected memory, as well as read-outs.

Fig. 7 presents the rules defining the relation $\Theta \xRightarrow{\bar{\alpha}} \Theta'$, which describe when a state Θ generates trace $\bar{\alpha}$ and results in state Θ' . The Tr^L traces of a program P is defined as follows: $\text{Tr}^L(P) = \{\bar{\alpha} \mid \exists \Theta'. \Theta_0(P) \xRightarrow{\bar{\alpha}} \Theta'\}$.

Example 3 below can now explain why protected code cannot write instructions in unprotected memory.

EXAMPLE 3 (WRITE-OUTS ARE NOT INSTRUCTIONS). *In the following example, protected programs can write instructions in unprotected memory. Recall that the protected memory partition spans from address 100 to 200, with one entry point at address 100.*

The following P_L and P_R set \mathbf{r}_0 to 20 and 10 respectively (line 1), then write the instruction `jmp \mathbf{r}_0` at address 20 and 10 respectively (line 2). Finally, they jump to the instruction they just wrote (line 3).

1	100	movi \mathbf{r}_0 20	1	100	movi \mathbf{r}_0 10
2	101	movs \mathbf{r}_0 "jmp \mathbf{r}_0 "	2	101	movs \mathbf{r}_0 "jmp \mathbf{r}_0 "
3	102	call \mathbf{r}_0	3	102	call \mathbf{r}_0

When \mathbf{r}_0 is set to 20 (resp. 10), the instruction `jmp \mathbf{r}_0` written at address 20 (resp. 10) will diverge. Thus, P_L and P_R are contextually equivalent, since no context can differentiate between them. However, P_L and P_R are trace inequivalent, since the following is a trace of P_L and not of P_R : $(\dots) \text{call } 100? \cdot \text{write}(20, \text{"jmp } \mathbf{r}_0") \cdot (\dots) \text{call } 20!$

Since we are interested in programs that do not extend the functionality of code in unprotected memory (and since these programs are the majority of assembly programs), protected code cannot write instructions in unprotected memory. Thus the programs of this example are not valid.

4.2 Tr^S : Changes to the Semantics

The Tr^S semantics presents different labels from Tr^L , which are driven by changes to the operational semantics of protected programs. These changes are formalised in **Fig. 8**; smaller changes to other rules are omitted. When the pro-

$$\begin{aligned} & \frac{(p, r, f, m, s) \xrightarrow{i} (p', r', f', m', s) \quad s \vdash \text{intJump}(p, p')}{(p, r, f, m, s) \xrightarrow{\tau_i} (p', r', f', m', s)} \quad (\text{Trace-internal}) \\ & \frac{(p, r, f, m, s) \xrightarrow{\tau_i} (p', r', f', m', s)}{(p, r, f, m, s) \xrightarrow{\tau_i} (p', r', f', m', s)} \quad (\text{Trace-internal-tick}) \\ & \frac{(p, r, f, m, s) \xrightarrow{i} (p', r', f', m', s) \quad s \vdash \text{protected}(p) \quad (p', r', f', m', s)^\perp}{(p, r, f, m, s) \xrightarrow{\surd} (p', r', f', m', s)} \quad (\text{Trace-writeout-accumulate}) \\ & \frac{s \vdash \text{intJump}(p, p+1) \quad m(p) = (\text{movs } \mathbf{r}_d \mathbf{r}_s) \quad s \vdash \text{unprotected}(r(\mathbf{r}_d)) \quad r(\mathbf{r}_s) \in \mathcal{V}}{(p+1, r, f, m, s) \xrightarrow{\delta!} (p', r', f', m', s)} \quad (\text{Trace-writeout-discard}) \\ & \frac{(p, r, f, m, s) \xrightarrow{\text{write}(r(\mathbf{r}_d), r(\mathbf{r}_s)).\delta!} (p', r', f', m', s)}{(p, r, f, m, s) \xrightarrow{\surd} (p', r', f', m', s)} \quad (\text{Trace-call}) \\ & \frac{s \vdash \text{intJump}(p, p+1) \quad m(p) = (\text{movs } \mathbf{r}_d \mathbf{r}_s) \quad s \vdash \text{unprotected}(r(\mathbf{r}_d))}{(p+1, r, f, m, s) \xrightarrow{\surd} (p', r', f', m', s)} \quad (\text{Trace-entryPoint}) \\ & \frac{(p, r, f, m, s) \xrightarrow{(r;f)\text{call } p?} (p, r, f, m, s)}{(p, r, f, m, s) \xrightarrow{(r;f)\text{ret } p?} (p, r, f, m, s)} \quad (\text{Trace-returnback}) \\ & \frac{s \vdash \text{exitJump}(p, p') \quad m(p) = (\text{call } p') \quad r' = r[\text{SP} \mapsto r(\text{SP}) + 1] \quad m' = m[r(\text{SP}) \mapsto p+1]}{(p, r, f, m, s) \xrightarrow{(r;f)\text{call } p!} (\text{unknown}, m', s)} \quad (\text{Trace-return}) \\ & \frac{p' = m(\text{SP}) \quad s \vdash \text{exitJump}(p, p') \quad m(p) = (\text{ret})}{(p, r, f, m, s) \xrightarrow{(r;f)\text{ret } p!} (\text{unknown}, m, s)} \quad (\text{Trace-trans}) \\ & \frac{}{\Theta \xRightarrow{\epsilon} \Theta} \quad (\text{Trace-refl}) \quad \frac{\Theta \xrightarrow{\tau_i} \Theta'}{\Theta \xRightarrow{\epsilon} \Theta'} \quad (\text{Trace-tau-i}) \quad \frac{\Theta \xRightarrow{\bar{\alpha}} \Theta''}{\Theta \xRightarrow{\bar{\alpha}\bar{\alpha}'} \Theta'} \quad (\text{Trace-action}) \quad \frac{\Theta \xrightarrow{\alpha} \Theta'}{\Theta \xRightarrow{\alpha} \Theta'} \end{aligned}$$

Figure 7: Rules of the Tr^L trace semantics.

gram counter jumps between the protected and the unprotected memory partitions, or vice-versa, flags are set to 0; in case of a `return`, all registers but \mathbf{R}_0 are also set to 0. Additionally, write-outs are prohibited.

Following are the labels of Tr^S ; we drop the greek letter notation and use latin letters in order to immediately differentiate between Tr^L and Tr^S . Flags do not appear in traces because they are always set to 0, as are all registers but \mathbf{R}_0 in case of a `return`. Write-outs are prohibited, so there are no labels that capture them.

$$L ::= a \mid \tau_i \quad a ::= \surd \mid g? \mid g! \quad g ::= \text{call } p(r) \mid \text{ret } p(r(\mathbf{r}_0))$$

Rules defining the relation $\Theta \xRightarrow{\bar{\alpha}} \Theta'$ for Tr^S are omitted. They are a subset of the ones defined in **Fig. 7**, with a slight modification to the syntax of generated labels. The Tr^S traces of a program P is defined as follows: $\text{Tr}^S(P) = \{\bar{a} \mid \exists \Theta. \Theta_0(P) \xRightarrow{\bar{a}} \Theta\}$.

4.3 Trace Equivalence

The notion of trace equivalence is presented generically for both trace semantics under consideration. Use $\text{Tr}(P)$ to

$$\begin{array}{c}
\text{(Aux-write-1')} \\
\frac{s \vdash \text{unprotected}(p) \quad s \vdash \text{unprotected}(a)}{s \vdash \text{writeAllowed}(p, a)} \\
\text{(Stack-out-to-in')} \\
\frac{s \vdash \text{entryJump}(p, p') \quad m' = m[\text{SP}_{\text{ext}} \mapsto r(\text{SP})] \quad r' = r[\text{SP} \mapsto m(\text{SP}_{\text{sec}})] \quad f' = [\text{ZF} \mapsto 0; \text{SF} \mapsto 0] \quad s \vdash \text{unprotected } r(\text{SP}) \quad s \vdash \text{protected } r'(\text{SP})}{p, r, f, m, s \vdash \text{setStack} \searrow p', r', f', m'} \\
\text{(Stack-in-to-out')} \\
\frac{s \vdash \text{exitJump}(p, p') \quad m' = m[\text{SP}_{\text{sec}} \mapsto r(\text{SP})] \quad r' = r[\text{SP} \mapsto m(\text{SP}_{\text{ext}})] \quad f' = [\text{ZF} \mapsto 0; \text{SF} \mapsto 0] \quad s \vdash \text{protected } r(\text{SP}) \quad s \vdash \text{unprotected } r'(\text{SP})}{p, r, f, m, s \vdash \text{setStack} \searrow p', r', f', m'} \\
\text{(Eval-return')} \\
\frac{m(p) = (\text{ret}) \quad p' = m(r(\text{SP})) \quad s \vdash \text{exitJump}(p, p') \quad r' = r[\text{SP} \mapsto r(\text{SP}) - 1; R_i \mapsto 0_{i=0..11}] \quad p, r', f, m, s \vdash \text{setStack} \searrow p', r'', f', m'}{(p, r, f, m, s) \rightarrow (p', r'', f', m', s)}
\end{array}$$

Figure 8: Changes to auxiliary functions and to the operational semantics for Tr^S .

indicate the traces of a program P , be it expressed through Tr^L or Tr^S .

Two programs P_1 and P_2 are trace-equivalent, denoted as $P_1 \simeq_T P_2$, if their traces are the same and they have the same memory descriptor.

DEFINITION 6 (TRACE EQUIVALENCE). $P_1 \simeq_T P_2$ if $\text{Tr}(P_1) = \text{Tr}(P_2)$ and $P_1 = (m_1, s)$ and $P_2 = (m_2, s)$.

Following are two examples of trace equivalent and inequivalent programs. For the sake of simplicity, we use the Tr^L semantics and indicate arbitrary values for registers and flags with notation (r, f) and an unprotected address with p .

EXAMPLE 4 (TRACES OF PREVIOUS EXAMPLES). *Code in*

Example 1 is not trace equivalent; the following trace is generated by P_L but not by P_R : $(r; f)\text{call } 100? \cdot (\dots, 41; f)\text{ret } p! \cdot \checkmark$. The code of *Example 2* is trace equivalent since the trace semantics of both P_L and P_R is a set whose sequences are concatenations of the following trace, each element of the sequence being parametrised on r and f : $(r; f)\text{call } 100? \cdot (r[R_0 \mapsto 0]; f)\text{ret } p!$

5. FULL ABSTRACTION THEOREM

This section presents the general proof strategy through which both Tr^L and Tr^S are proven to be fully abstract w.r.t. the corresponding operational semantics and discusses the benefits of full abstraction of the trace semantics.

A fully abstract trace semantics is both sound and complete with respect to the operational semantics. Soundness states that the trace semantics captures all details of the operational semantics. Thus, for all contexts, two trace equivalent programs cannot be told apart. The universal quantification over contexts makes this more difficult to prove. Completeness states that abstractions of the trace semantics are preserved in the operational semantics. This is achieved by proving the contrapositive of the completeness statement. For this, the proof consists of devising an algorithm that constructs a program, a “witness”, that tells two trace inequivalent programs apart [2, 6, 14]. Since the language is low-level and untyped, this is not particularly complicated.

Full abstraction of trace semantics is formally stated as: $P_1 \simeq_T P_2 \iff P_1 \simeq P_2$; its proof is split in two cases, one for each direction of the co-implication.

THEOREM 1 (SOUNDNESS). $P_1 \simeq_T P_2 \Rightarrow P_1 \simeq P_2$.

Call the *interface* of a state its registers, flags and unprotected memory. Two states Ω_1 and Ω_2 have the same interface, denoted as $\Omega_1 \doteq \Omega_2$, if they have the same registers, flags and unprotected memory. Formally, $\Omega_1 \doteq \Omega_2$ if $\Omega_1 = (p_1, r, f, m_1, s_1)$ and $\Omega_2 = (p_2, r, f, m_2, s_2)$ and $m_{\text{ext}}(m_1, s_1) = m_{\text{ext}}(m_2, s_2)$. Given $\Omega = (p, r, f, m, s)$, redefine $[\Omega]$ to be state $\Theta = (p, r, f, m_{\text{sec}}(m, s), s)$ if $s \vdash \text{protected}(p)$ and $(\text{unknown}, m, s)$ otherwise.

The proof of **Theorem 1** states that an unprotected program interacting with P_1 cannot distinguish it from P_2 . This is because both programs offer the same interface to the unprotected program. So this proof depends on an interface-preservation lemma (**Lemma 1**) which must be proven for each trace semantics since it depends on the labels of each trace semantics. **Lemma 1** enunciates that two states with the same interface still have the same interface after they perform the same observable action. Thus unprotected programs do not see differences, in terms of flags, registers and unprotected memory, between P_1 and P_2 .

LEMMA 1 (INTERFACE PRESERVATION). If $\Theta_1 \xrightarrow{\bar{\alpha}} \Theta'_1$ and $\Theta_1 = [\Omega_1]$ and $\Omega_1 \rightarrow^* \Omega'_1$ and $\Theta'_1 = [\Omega'_1]$ and $\Theta_2 \xrightarrow{\bar{\alpha}} \Theta'_2$ and $\Theta_2 = [\Omega_2]$ and $\Omega_2 \rightarrow^* \Omega'_2$ and $\Theta'_2 = [\Omega'_2]$ and $\Omega_1 \doteq \Omega_2$ then $\Omega'_1 \doteq \Omega'_2$.

THEOREM 2 (COMPLETENESS). $P_1 \simeq P_2 \Rightarrow P_1 \simeq_T P_2$.

Completeness is equivalently stated as: $P_1 \not\simeq_T P_2 \Rightarrow P_1 \not\simeq P_2$. This is proven by devising an algorithm that takes as input two different traces $\bar{\alpha}_1$ and $\bar{\alpha}_2$ and the two programs P_1 and P_2 generating them and outputs a program P that interacts with P_1 and P_2 and is able to differentiate between them. The two different traces are generated as follows. Since $P_1 \not\simeq_T P_2$, we have that $\text{Tr}(P_1) \neq \text{Tr}(P_2)$, thus there exists a trace $\bar{\alpha}$ that belongs to either only $\text{Tr}(P_1)$ or only $\text{Tr}(P_2)$. Assume wlog that $\bar{\alpha} \in \text{Tr}(P_1)$. The trace $\bar{\alpha}$ can be split in two parts $\bar{\alpha}_s$ and $\bar{\alpha}_d$ such that $\bar{\alpha} = \bar{\alpha}_s \bar{\alpha}_d$, and so that there exists a trace $\bar{\alpha}' \in \text{Tr}(P_2)$ that can be split in two parts $\bar{\alpha}'_s$ and $\bar{\alpha}'_d$ such that $\bar{\alpha}' = \bar{\alpha}'_s \bar{\alpha}'_d$ and $\bar{\alpha}_d \neq \bar{\alpha}'_d$. Trace $\bar{\alpha}'$ always exists, it could be an empty trace, it could be composed by an empty $\bar{\alpha}_s$ and, possibly, by an empty $\bar{\alpha}'_d$. The traces input for the algorithm are $\bar{\alpha}_1 = \bar{\alpha}_s \bar{\alpha}_d$ and $\bar{\alpha}_2 = \bar{\alpha}_s \bar{\alpha}'_d$.

The interested reader is referred to the companion technical report for proofs [13]. This general proof strategy is presented for both **Theorem 1** and **Theorem 2**. The generalised approach is tailored to each semantics only in the relatively simple proof of **Lemma 1**. Since **Theorem 1** and **2** hold for both Tr^L and Tr^S , both semantics are fully abstract w.r.t. the corresponding operational semantics.

5.1 Benefits of the Trace Semantics

Let us now highlight two of the benefits of fully abstract trace semantics.

Without the trace semantics, the capabilities of an attacker towards protected code are expressed as sequences of instructions, which can be difficult to reason about. With the trace semantics, the capabilities of that attacker are captured via the simple notion of traces. The full abstraction property ensures that traces express precisely all the capabilities of an attacker, without leaving any out.

The second benefit is in the field of secure compilation. Given two programs C_1 and C_2 in a language and indicate their compilation to an assembly language with C_1^\downarrow and C_2^\downarrow respectively. Proving the compilation scheme secure is formally stated as $C_1 \simeq C_2 \iff C_1^\downarrow \simeq C_2^\downarrow$ [1]. The more complex direction of this proof is $C_1 \simeq C_2 \Rightarrow C_1^\downarrow \simeq C_2^\downarrow$, but it can be simplified by adopting a fully abstract trace semantics for the assembly language. That proof is simpler in this way than by adopting the notion of contextual equivalence and performing induction on all possible low-level contexts. The complex direction becomes $C_1 \simeq C_2 \Rightarrow C_1^\downarrow \simeq_{\text{T}} C_2^\downarrow$, whose contrapositive is $C_1^\downarrow \not\simeq_{\text{T}} C_2^\downarrow \Rightarrow C_1 \not\simeq C_2$. This can be proven similarly to [Theorem 2](#), as Agten *et al.* [2] and Patrignani *et al.* [14] did. Both implicitly use the second trace semantics where the compiler adds code that enforces the changes to the operational semantics.

6. RELATED WORK

Full abstraction has been largely studied as a way to state the correctness of a denotational semantics with respect to an operational one [15]. This technique has been adopted for different programming languages paradigms, such as the λ -calculus [10] and the π -calculus [5].

Fine grained, program counter-based memory access control mechanisms have been implemented in several software [8, 17, 18, 19] and hardware forms [4, 12] and recently in the Intel SGX processor [9]. From the theoretical point of view, assembly languages extended with these protection mechanisms have been recently studied as target languages for secure compilation schemes [2, 14]. The language and trace semantics of this paper are inspired by those works. Besides elaborating on details of their formalisation, this work provides the first proof of full abstraction of the trace semantics.

A different research area studies logics for assembly languages: Hoare logics [16] or separation logics [7]. Jensen *et al.* [7] present a summary of the most recent advances in the latter. That research area focusses on providing reasoning facilities for assembly code, while this paper focusses on reasoning on the security of assembly code.

A different protection mechanism that could be employed at the assembly level is a typed assembly language [11]. To the best of the authors' knowledge, no fully abstract trace semantics has been provided for such languages.

7. CONCLUSION AND FUTURE WORK

This paper studied the characterisation of the behaviour of isolated programs. To this extent, it formalised an assembly language extended with a fine grained, program counter-based memory access control mechanism. Then, it provided two different trace semantics for that language and a general proof strategy where both semantics are proven to be fully abstract. These semantics model the capabilities of attackers that inject malicious assembly code and they simplify proving secure compilation to the assembly language.

Providing a fully abstract trace semantics for a machine with multiple instances of an isolation mechanism or with multiple cores seem natural extensions to this work. The latter seems crucial in order to provide a secure compiler for concurrent programs to machines using the protection mechanism presented.

Acknowledgements. The authors would like to thank Tarmo Uustalu and the reviewers for useful feedback on an earlier draft.

8. REFERENCES

- [1] M. Abadi. Protection in programming-language translations. In *Secure Internet programming*, pages 19–34. Springer-Verlag, 1999.
- [2] P. Agten, R. Strackx, B. Jacobs, and F. Piessens. Secure compilation to modern processors. In *CSF '12*, pages 171 – 185. IEEE, 2012.
- [3] P.-L. Curien. Definability and full abstraction. *Electr. Notes Theor. Comput. Sci.*, 172:301–310, 2007.
- [4] K. Eldefrawy, A. Francillon, D. Perito, and G. Tsudik. SMART: Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust. In *NDSS'12*.
- [5] A. Jeffrey and J. Rathke. Full abstraction for polymorphic pi-calculus. In *FOSSACS'05*, pages 266–281. Springer-Verlag, 2005.
- [6] A. Jeffrey and J. Rathke. A fully abstract may testing semantics for concurrent objects. *Theor. Comput. Sci.*, 338(1-3):17–63, 2005.
- [7] J. B. Jensen, N. Benton, and A. Kennedy. High-level separation logic for low-level code. *SIGPLAN Not.*, 48(1):301–314, Jan. 2013.
- [8] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: an execution infrastructure for TCB minimization. *SIGOPS Oper. Syst. Rev.*, 42(4):315–328, 2008.
- [9] F. McKeen *et al.* Innovative instructions and software model for isolated execution. In *HASP '13*, pages 10:1–10:1, New York, NY, USA, 2013. ACM.
- [10] R. Milner. Fully abstract models of typed λ -calculi. *Theor. Comput. Sci.*, 4(1):1–22, 1977.
- [11] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21(3):527–568, May 1999.
- [12] J. Noorman *et al.* Sancus: Low-cost trustworthy extensible networked devices with a zero-software Trusted Computing Base. In *Proceedings of the 22nd USENIX conference on Security symposium*, 2013.
- [13] M. Patrignani and D. Clarke. Fully Abstract Trace Semantics of Low-level Protection Mechanisms – Extended Version. CW Reports CW 651, Dept. of Computer Science, K.U.Leuven, November 2012.
- [14] M. Patrignani, D. Clarke, and F. Piessens. Secure Compilation of Object-Oriented Components to Protected Module Architectures. In *(APLAS'13)*, volume 8301 of *LNCS*, pages 176–191, 2013.
- [15] G. Plotkin. LCF considered as a programming language. *Theor. Comput. Science*, 5:223–255, 1977.
- [16] A. Saabas and T. Uustalu. A compositional natural semantics and Hoare logic for low-level languages. *Electr. Notes Theor. Comput. Sci.*, 156:151–168, 2006.
- [17] L. Singaravelu, C. Pu, H. Härtig, and C. Helmuth. Reducing TCB complexity for security-sensitive applications: three case studies. *SIGOPS Oper. Syst. Rev.*, 40(4):161–174, 2006.
- [18] R. Strackx and F. Piessens. Fides: Selectively hardening software application components against kernel-level or process-level malware. In *CCS 2012*, pages 2–13. ACM Press, October 2012.
- [19] R. Strackx, F. Piessens, and B. Preneel. Efficient isolation of trusted subsystems in embedded systems. In *SecureComm*, pages 344–361, 2010.